Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim

DOPSY
group
Labor für Verteilte Systeme
Distributed Systems Lab

Abschlussbericht BMBF

AQUAS

HSRM

Stand: 23. Januar 2020

Inhaltsverzeichnis

1 Kurzdarstellung **3**
- 1.1 Aufgabenstellung . . . 3
- 1.2 Voraussetzungen . . . 3
- 1.3 Planung und Ablauf des Vorhabens . . . 3
- 1.4 Gegebener wissenschaftlicher und technischer Stand . . . 3
  - 1.4.1 Rechercheergebnisse . . . 3
  - 1.4.2 Benutzte Verfahren und Schutzrechte . . . 3
  - 1.4.3 Benutzte Informations- und Dokumentationsdienste . . . 3
- 1.5 Zusammenarbeit mit anderen Stellen . . . 3

2 Eingehende Darstellung **4**
- 2.1 Mit der Zuwendung erzielte Ergebnisse . . . 4
  - 2.1.1 Überblick . . . 4
  - 2.1.2 Methode . . . 4
  - 2.1.3 Geschichteter Kernel . . . 12
  - 2.1.4 Evaluation . . . 12
  - 2.1.5 AQUAS Kontext . . . 13
- 2.2 Wichtigste Positionen des zahlenmäßigen Nachweises . . . 14
- 2.3 Notwendigkeit und Angemessenheit der geleisteten Arbeit . . . 15
- 2.4 Voraussichtlicher Nutzen . . . 15
- 2.5 Fortschritt bei anderen Stellen . . . 15
- 2.6 Erfolgte und geplante Veröffentlichungen . . . 15

3 Erfolgskontrollbericht **16**
- 3.1 Beitrag des Ergebnisses zu den förderpolitischen Zielen des Förderprogramms (Steigerung der Verbund-/Drittmittelfähigkeit . . . 16
- 3.2 Wissenschaftlich-technische Ergebnisse des Vorhabens, erreichte Nebenergebnisse und gesammelte Erfahrungen . . . 16
- 3.3 Fortschreibung des Verwertungsplans . . . 16
  - 3.3.1 Erfindungen/Schutzrechtsanmeldungen . . . 16
  - 3.3.2 Wirtschaftliche Erfolgsaussichten nach Projektende . . . 16
  - 3.3.3 Wissenschaftliche und/oder technische Erfolgsaussichten . . . 16
  - 3.3.4 Wissenschaftliche und wirtschaftliche Anschlussfähigkeit . . . 16
- 3.4 Arbeiten, die zu keiner Lösung geführt haben . . . 16
- 3.5 Präsentationsmöglichkeiten für mögliche Nutzer . . . 16
- 3.6 Einhaltung der Ausgaben- und Zeitplanung . . . 16

Literatur **17**

1 Kurzdarstellung

1.1 Aufgabenstellung

(1 - 2 S)

1.2 Voraussetzungen

(1 S)

1.3 Planung und Ablauf des Vorhabens

(2 S)

- Änderungen
  - Organisation Arbeitspakete

1.4 Gegebener wissenschaftlicher und technischer Stand

(insg. 4 S)

1.4.1 Rechercheergebnisse

1.4.2 Benutzte Verfahren und Schutzrechte

1.4.3 Benutzte Informations- und Dokumentationsdienste

1.5 Zusammenarbeit mit anderen Stellen

(3 - 4 S)

2 Eingehende Darstellung

2.1 Mit der Zuwendung erzielte Ergebnisse

2.1.1 Überblick

(2 S)

- Versprochenes aus dem Antrag
- Wie Abänderung zu BMBF Antrag durch Einbettung in Kontext AQUAS -> Kaiser
- Grafik?

2.1.2 Methode

2.1.2.1 Überblick

(3 S)

Zuverlässige Systeme erfordern ein hohes Maß an Vertrauen in ihre Software. In der Regel wird dieses Vertrauen durch Zertifizierung hergestellt. Je nach Einsatzgebiet existieren unterschiedliche Standards und Normen, welche die Software hinsichtlich eines Safety Integrity Levels (SIL) [IEC] zertifizieren. Die SIL Stufe gibt dabei die notwendigen Kriterien hinsichtlich Testabdeckung oder formaler Verifikation des Quellcodes vor. Während für die Validierung durch Testen bereits viele einschlägige Methoden und Praxiswissen in der Industrie existiert, ist die Nutzung formaler Verifikation bisher viel geringer. Ziel des Teilvorhabens der Hochschule RheinMain im Projekt AQUAS war es, den Aufwand für die Entwicklung formal verifizierter Software zu reduzieren und den Zertifizierungsaufwand dadurch zu minimieren. Auf diese Weise soll formale Verifikation in der Software-Entwicklung zugänglicher gemacht werden.

Die im Projekt entwickelte Methode wurde am Beispiel eines Mikrokernels entwickelt und evaluiert. Dafür wurde der geschichtete Aufbau eines Mikrokernels ausgenutzt, der ein inkrementelles Vorgehen durch einen hierarchischen Ansatz, wie in [LNR+80] beschrieben, ermöglicht. (siehe Abb. 2.1)

Der Mikrokernel wird dabei in folgende Schichten aufgeteilt:

- **Single-Tasking System**, ein Mikrokernel ohne Nebenläufigkeit
- **Multi-Tasking System without MPU**, ein nebenläufiger Mikrokernel
- **Multi-Tasking System with MPU**, ein nebenläufiger Mikrokernel mit Speicherschutz durch MPU

- **Multi-Tasking System with MMU**, ein nebenläufiger Mikrokernel mit virtuellem Speicher durch MMU

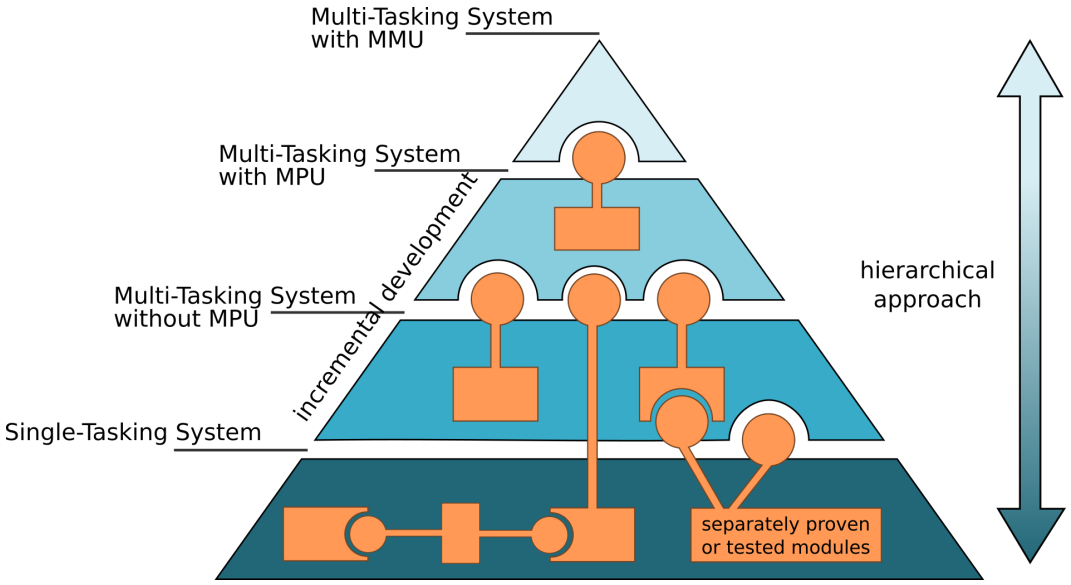

Abbildung 2.1: Inkrementelle Entwicklung durch hierarchischen Ansatz

Die hierarchische Schichtung ermöglicht nicht nur eine inkrementelle Entwicklung der Software, sondern auch der Beweise. Wie in Abb. 2.1 dargestellt, besteht jede Schicht aus unterschiedlichen Modulen, die von der aktuellen oder den höheren Schichten genutzt werden. Jedes Modul kann separat bewiesen oder getestet werden, so dass die darauf aufbauenden Schichten die bewiesenen Module als Bausteine verwenden können.

Die formale Verifikation des Quellcodes wird durch automatisierte bzw. semi-automatisierte Werkzeuge wie SPARK 2014 [spaa] oder Coq [coq] unterstützt. SPARK 2014 ist eine Programmiersprache, welche auf der Sprache Ada 2012 basiert und durch die gleichnamige Beweisumgebung eine automatisierte Verifikation von SPARK Quellcode ermöglicht. Durch ihre strikte Typisierung und die Möglichkeit, Contracts als Aspekte zu definieren, eignet sich Ada besonders gut für die durchführung formaler Beweise. Zusätzlich schränkt SPARK Ada ein und verbietet nicht-verifizierbare Sprachkonstrukte wie z.B. dynamische Speicherallokation, Zeiger, Rekursion usw. Durch die Nutzung von SPARK ist es möglich, funktionalen Quellcode und Verifikationscode zusammen zu halten und somit Übertragungsfehler zu reduzieren.

In der Praxis sind die meisten Systeme jedoch entweder zu komplex oder besitzen Komponenten, deren Zustand erst zur Laufzeit bekannt wird, so dass eine vollständige statische Verifikation nicht möglich ist. Dies gilt beispielsweise für lose gekoppelte verteilte Systeme, wo Daten während der Übertragung ihre Typisierung verlieren oder Systemen, die von Umgebungszuständen oder Benutzereingaben abhängen, die sich zur Laufzeit dynamisch ändern können. Um auch Systeme, die für statische Verifikation nicht geeignet sind, verifizieren zu können, werden die dynamischen Komponenten des Systems zur Laufzeit getestet. Diese Vorgehensweise gehört zur allgemeineren Methode der Runtime Verification [ML08].

<ToDo: Timing Verifikation und Coq>

2.1.2.2 Funktionale Verifikation

(5 S)

Die Methode zur funktionalen Verifikation gliedert ich in die Subthemen der statischen Verifikation, dynamischen Verifikation und einer Kombination beider Vorgehensweisen. Im Folgenden werden die drei Ansätze erläutert.

Statische Verifikation

Statische formale Verifikation befasst sich mit dem formalen Beweis von Quellcode durch eine statische Analyse zum Entwicklungszeitpunkt. Im Rahmen des AQUAS Projekts wurde die Vorgehensweise der mathematischen Deduktion von prädikatenlogischen Aussagen gewählt, um die formale Korrektheit von Quellcode nachzuweisen. Als Werkzeug wurde die automatische Beweisumgebung und Programmiersprache SPARK 2014 [spaa] eingesetz. SPARK ermöglicht die Definition von Contracts für Subprogramme, Klassen und Pakete. Sie werden von SPARK in die Kategorien Dependency Contracts, Preconditions und Postconditions unterteilt [spab] und stellen eine Vereinbarung zwischen dem Aufrufer und der Schnittstelle dar. Dabei beschreiben Preconditions die Bedingungen, die der Aufrufer erfüllen muss, Postconditions beschreiben den Zustand nach dem Aufruf der Schnittstelle und Dependency Contracts die Abhängigkeiten zwischen den Daten bzw. den Datenfluss und dienen somit als Hilfsmittel bei der Analyse der Pre- und Postconditions.

Der geschichtete Aufbau des entwickelten Mikrokernels ermöglicht den Einsatz der hierarchischen Entwicklungsmethode (HDM) [LNR+80]. Dieser Ansatz wurde insbesondere auf das Schnittstellendesign übertragen. Das hierarchische Design der Schnittstelle ermöglicht es auch die Contracts hierarchisch zu definieren. Die Prädikate der unteren Schnittstellen verallgemeinern dabei die spezialisierten Prädikate der höheren Schichten, so dass sich diese spezialisierten Prädikate aus den allgemeineren ableiten lassen. Das Beispiel einer Queue, die auf der Implementierung einer doppelt verketteten Liste basiert soll hierfür die Vorgehensweise veranschaulichen (Abb. 2.2).

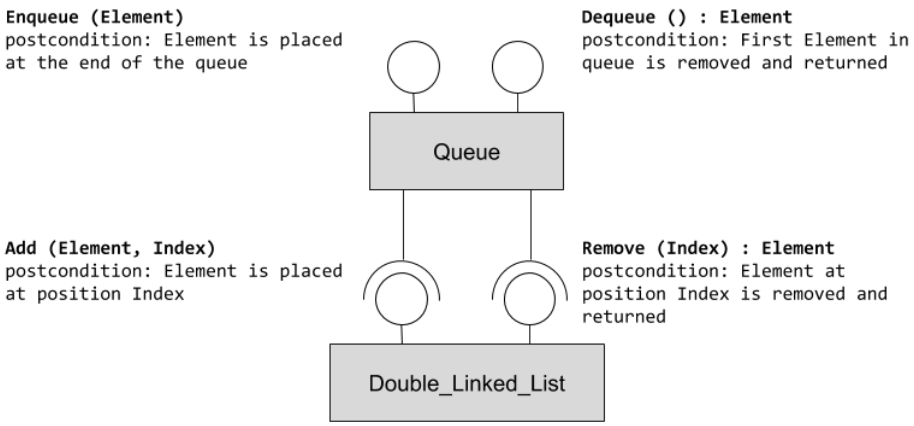

Abbildung 2.2: Verfeinerung allgemeiner Prädikate durch hierarchischen Ansatz

Zum Hinzufügen neuer Elemente bietet die doppelt verkettete Liste die Schnittstelle `Add` an, die mit einem `Element` und einem `Index` aufgerufen wird. Die Postcondition von `Add` garantiert, dass `Element` nach der Ausführung an der Position `Index` in der Liste gespeichert wurde. Queue bietet die Schnittstelle `Enqueue` an, die mit einem `Element` aufgerufen wird. Dabei garantiert die Postcondition von `Enqueue`, dass das übergebene `Element` am Ende der Queue gespeichert wird. `Enqueue` wird hier mit Hilfe von `Add` implementiert, wobei als `Index` des Listenende verwendet wird. Die Postcondition von `Enqueue` gilt somit dann, wenn die Postcondition von `Add` gilt. `Dequeue` und `Remove` verhalten sich analog dazu.

Um diese Eigenwschaft ausnutzen zu können sieht die hier entwickelte Methode deshalb vor ein System in SysML/OCL modellieren zu können. Da zur Modellierung des Systems und dessen Constraints spezifisches Domänenwissen benötigt wird, sind SysML/OCL auf Grund ihrer weiten Verbreitung als Industriestandard hierfür besonders gut geeignet. Die Mächtigkeit von SysML/OCL ermöglicht die Modellierung von mehr als nur den funktionalen Eigenschaften, so dass zur weiteren Analyse des Modells durch Transformation unterschiedliche Sichten auf das System geschaffen werden können (vgl. Abb 2.3).

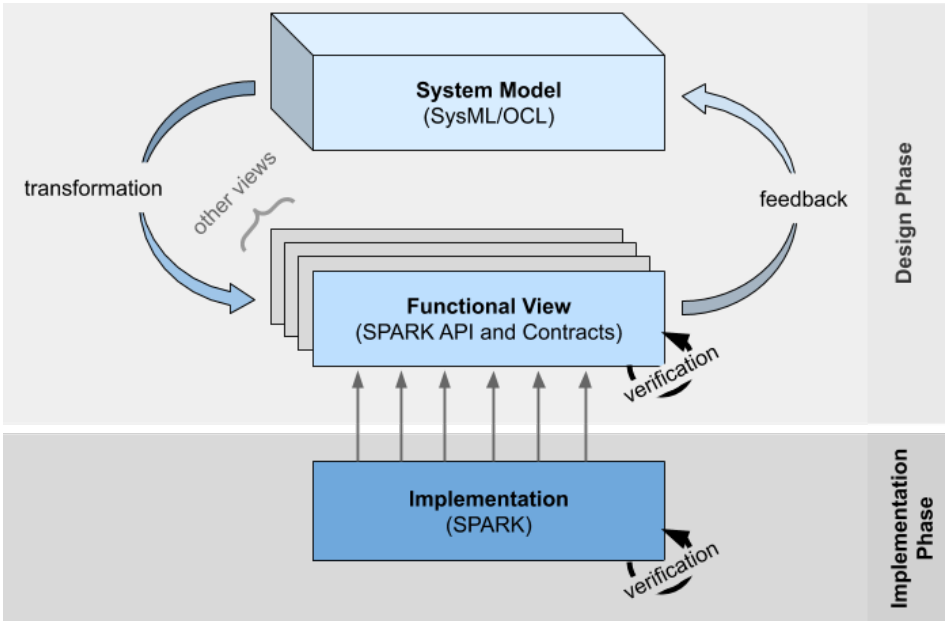

Abbildung 2.3: Transformation und funktionale Verifikation des Systemmodells

Die Methode der funktionalen Verifikation betrachtet die funktionale Sicht. Dabei wird die SysML-Spezifikation in ein SPARK API transformiert und OCL-Constraints in dazugehörige SPARK Contracts. Dadurch wird es möglich die Schnittstellensemantik bereits zum Design-Zeitpunkt zu verifizieren und somit frühzeitig Feedback zurückführen zu können. Die Verifikation der Implementierung kann anschließend separat zum Implementierungszeitpunkt geschehen. Die Korrektheit der Interface-Semantik ist damit zwar von der Korrektheit der Implementierung anhängig, jedoch nicht von einer bestimmten Implementierung.

Dynamische Verifikation

In der Regel sind reale Systeme zu komplex oder weisen Zustände auf, die erst zur Laufzeit bekannt sind, so dass eine vollständige statische Verifikation solcher Systeme nicht möglich ist. Ein häufig verfolgter Ansatz ist es dann nicht verifizierten Code zu testen. Durch den hierarchischen Ansatz der Methode ist es möglich, einzelne Prädikate und Teilprädikate für das Testen zu identifizieren. Da die Contracts die Anforderungen des Systems bereits formal spezifizieren, können diese als Testorakel verwendet werden. Die nicht-verifizierten Prädikate können also ausgeführt und das tatsächliche Verhalten zur Laufzeit beobachtet und bewertet werden. Das Vorgehen des Testens beschränkt sich allerdings rein auf den Entwicklungszeitraum. Die Ausführung der Prädikate findet somit keinen Einzug in die ausgelieferte Software. Damit entsteht das, für den Bereich des Testens, übliche Problem der Testabdeckung. Dieses Problem kann gelöst werden, in dem der Ansatz Verifikation und Testen zu kombinieren, in der Methode der Runtime Verification [ML08] verallgemeinert wird.

Runtime Verification beschreibt eine Methode der dynamischen Verifikation bei der Monitore die zu verifizierenden Eigenschaften des Systems zur Laufzeit überwachen. In der Vergangenheit wurde nur die reine Überwachung als Runtime Verification betrachtet, doch diese Sichtweise hat sich geändert, so dass heutzutage auch die Adaption des Systems als Folge der Nichteinhaltung überwachter Eigenschaften als Teil der Runtime Verification angesehen wird [ML08].

Die Kombination statischer und dynamischer Verifikation

Die hier entwickelte Methode kombiniert die Ansätze der statischen und dynamischen Verifikation. Sie gliedert sich hierfür in eine Vorgehensweise zum Designzeitpunkt und eine Laufzeitarchitektur für Runtime Verification.

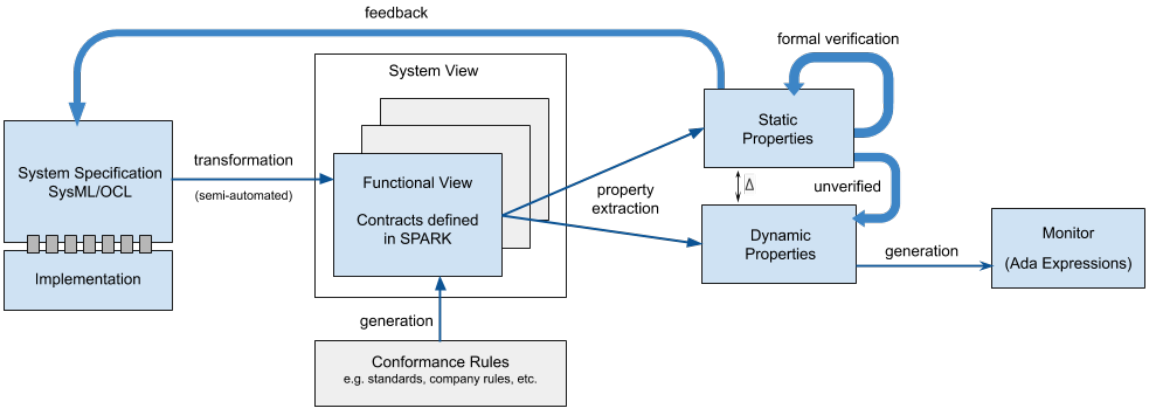

Abbildung 2.4: Methode zum Designzeitpunkt

Zum Designzeitpunkt wird, wie bereits erläutert, von einem System Modell in SysML/OCL ausgegangen, welches dann in eine funktionale Sicht transformiert wird (siehe Abb. 2.4). Da SysML/OCL maschinenverarbeitbar ist, soll dieser Schritt semi-automatisch erfolgen. Die Transformation soll hier nach SPARK erfolgen, jedoch könnte an dieser Stelle auch jede

andere Technologie verwendet werden, die das Formalisieren von Contracts und deren Verifikation ermöglicht. Da SysML/OCL eine viel größere Mächtigkeit als SPARK aufweist, ist es empfehlenswert, diese durch ein entsprechendes Profil einzuschränken. Neben den in OCL definierten Anforderung, können weitere Anforderungen durch externe Vorschriften wie z.B. Standards eingehen, die ebenfalls als SPARK Contracts definiert werden. Die Gesamtmenge aller definierten Contracts wird dann unterteilt in Eigenschaften, die sich statisch verifizieren lassen und solche, die dynamisch überprüft werden müssen. Die Verifikation der statisch Eigenschaften erfolgt für die Schnittstellensemantik zum Designzeitpunkt und für die konkrete Implementierung zum Implementierungszeitpunkt. Prädikate, die sich dabei als nicht-verifizierbar erweisen, werden gemeinsam mit den anderen dynamischen Eigenschaften zur Laufzeit verifiziert. Dafür werden aus den Prädikaten und Teilprädikaten der SPARK Contracts Monitore generiert. Da Prädikate in SPARK bereits formal als Expressions definiert werden, können diese als Ada Expression Functions zur Überwachung von den Monitoren ausgeführt werden.

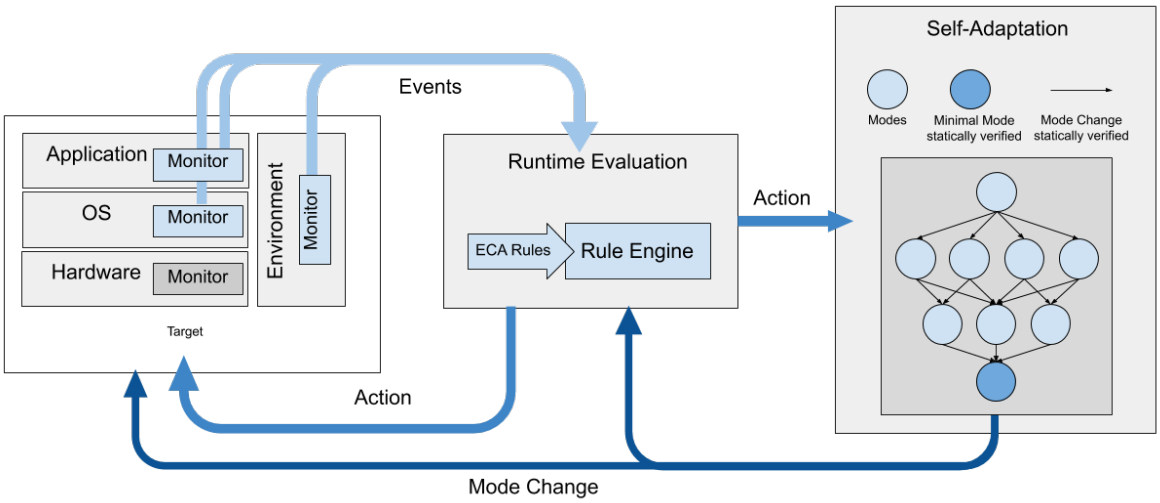

Abbildung 2.5: Laufzeitarchitektur für Runtime Verification

Die Laufzeitarchitektur sieht vor, dass die zuvor generierten Monitore sich auf unterschiedlichen Schichten des Zielsystems befinden und dieses überwachen (siehe Abb. 2.5). Somit können beispielsweise Preconditions deren Einhaltung statisch nicht nachgewiesen werden kann, zur Laufzeit überprüft werden. Neben den System- und Applikationsmonitoren, spielen Umgebungsmonitore für Systeme, deren Zustände von Umgebungseinflüssen abhängen, eine besonders wichtige Rolle. Die hier entwickelte Methode befasst sich jedoch nicht mit der Generierung und überwachung von Hardwaremonitoren. Die Verletzung überwachter Eigenschaften löst ein Ereignis aus. Diese Ereignisse werden durch eine Rule Engine auf basis von Event Condition Action (ECA) Rules ausgewertet und führen entweder zu einer einfachen Rekonfiguration oder einer Selbstadaption des Systems.

Die Selbstadaption bietet nicht nur die Möglichkeit von Functional Degradation, sondern auch einen garantierten Wechsel in einen sicheren, verifizierten Zustand im Notfall. Das System sieht dafür unterschiedliche Modi vor in denen es betrieben werden kann und die mehr oder weniger Funktionalität aufweisen. Modi sind in einem Graphen mit definierten

übergängen zwischen den verschiedenen Modi angeordnet. Der Minimal Mode bezeichnet dabei den sicheren Zustand des System und ist in seiner Korrektheit statisch verifiziert und über ein oder mehrere Zustandsübergänge aus jedem Modus erreichbar. Zusätzlich zu dem Minimal Mode muss auch der der Algorithmus zum Modewechsel statisch verifiziert sein, damit die Selbstadaption einen Wechsel aus jedem Modus in den Minimal Mode garantieren kann.

Die beschriebene Laufzeitarchitektur wurde während der Laufzeit des Projektes nicht implementiert und die Designmethode bisher nur teilweise getestet.

Funktionale Verifikation mit Coq

⇒ Almeroth

- Analyse Coq
- Das Betreibssystem in AQUAS wurde irgendwann ausgetauscht. Das sollte im unten stehenden Abschnitt ersetzt werden.

Der Coq Beweis-Assistent benutzt grundelegende Prinzipen der Logik und der fuktionalen Programmierung [PdAC+19] für die semi-interaktive maschinen-überprüfte Beweisführung [coq].

Wegen des Erfolges in der Echt-Zeit-Verarbeitungs-Community durch die Formale Vertifikation eines Plaungsverfahrens (vgl. [CSB16]) und die formale Vertifikation des Kernes eines Einzelkern-Betriebsystems [GLL+19] soll eine Funktionale Vertifikation des Planunsgverfahrnes aus diesem Projekt durchgeführt werden. Diese Funktionale Vertifikation sollen mittels des sogenannten PROSA-Framewokrs, einer Open-Source Coq-Bibliothek, geschehen (vgl. [PdAC+19] [PRO]). PROSA ist das Akronym für den Titel des Projektes *Formally Proven Schedulability Analysis*. Die Vorteile der Integration der Formalen Planbarkeitsanalyse mittels PROSA sind dass es keinen *Runntime-Overhead* und zu keinem *Performace Overhead* des Planungsverfahrens kommt [PRO].

Den großen Erfolg, den PROSA hatte ist drauf zurück zu führen, dass es ein Framework bietet, das Plangsverfahren maschienen-überprüfbar macht [PRO]. Aus historischen Günden [PRO] (vgl. [THW94][BLDB06],) ist dies in der Planungsanalyse erstrebenswert. Außerdem soll das Verwenden des PROSA Framewokes den Einstieg in die semi-automatische Beweisführung einfach gestallten und die Beweise sollen für einen Leser ohne großes Mathematisches Vorwissen lesbar sein. *State-of-the-art* Planungs-Analysen liegen in PROSA schon in maschinen-überprüfbarer Form vor. Das heisst, dass auch ohne alle Zwischenergebnisse per Hand zu beweisen ein beliebiges Plaungsverfahren maschienen-überpüft werden kann [PRO].

Computergestützte Beweisverfahren wie sie mit Beweis-Assistenten wie *Isabelle* (s. [Isa]) oder Coq durchgeführt werden, finden in der Softwareenwicklung Verwendung um genaue Aussagen über Programme zu formulieren. Beweis-Assistenten wenden grundlegende Werkzeuge der Logik an um Logisches Schließen über Programme zu implementieren [PdAC+19]. Wegen Coqs *algebraischen Daten Typen* und *pattern matching* kann mithilfe des Coq-Beweisassistenten ein mächtigere Werkzeug zur Mathemaischen Deduktion als Prädikatenlogik wie bei der Verwendung der Beweisumgebung SPARK angewendet werden (s. Abschnitt 2.1.2.2).

Insbesonder wurde mittels PROSA ein dynamisches System und eine sogenannte *response time analysis* auf einem identischen multicore Planungsverfahren erfolgreich validiert [PRO] (vgl. Paragraph 2.1.2.2). Dabei sei darauf hingeweisen, dass für eine Planbarkeitsanalys eines Multiprozessorsystems diese Annahme keines falls trivial ist. Insbesondere stellt sich die Frage nach der Modellierung des Zeitverhaltens eines dynamischen Systems. Durch die Coq Erweiterung SSREFELCT [ssr] wird in PROSA Zeit aus mathematischer Sicht diskret dargestellt. Zeit ist ein Alias auf die natürlichen Zahlen $\mathbb{N}$ [PRO].

Die dem Coq-Beweisassistenten unterliegende formale Programmiersprache heißt Gallina [PdAC+19]. Diese biet drei Untertypen an [PdAC+19]. Multiple Programmierparadigmen können mittels dieser Sprache implementiert werden unter anderem auch Rekursion und Schleifen [PdAC+19].

Um mit Coq eine Planbarkeitsanalyse durchzuführen ist es notwendig alle Attribute des in Marron verwenden Plaungsverfahren zu formalisieren. Die Systemkomponenten müssen komplett beschrieben werden. Es wurde damit begonnen die Anforderungen des EDF-Schedulers eines anderen Betriebssystemes [Kai09] aufzuschreiben als ein Wirksamkeitsbeweis.

- Alles, was jetzt noch produziert wird hier hinein schreiben

Dies führt zu einer Planbarkeitsanaylse um die erstebenswerten Attrribute des in Marron verwendeten Planungsverfahrens zu formulieren. Das Wasserfallmodel für eine Entwicklung wird in diesem Entwicklungsprozess verwendet, da jede rekursive Arbeitsweise zu einer Neudefinition der formal zu vertifizierenden Attribute führt.

In den Arbeiten [GLL+19] und [PRO] wurde auch auf diese Art gearbeittel, da die Planungsverfahren nicht während der Vertifizierung entwickelt wurden.

Das Betriebssystem RT-CertiKOS, welches von der Yale FLINT-Gruppe entwickelt wurde, ist eine Echtzeit Erweiterung des single-core sequential CertiKOS Betriebssystems. Die funktionale Korrektheit von RT-CertiKOS konnte mit dem Coq Beweis-Assistenten gezeigt werden [GLL+19]. Es soll darauf hingeweisen werden, dass dabei mehrer Abstraktionsebenen verwendet wurden und bei der Vertifikation PROSA nicht benutzt wurde [GLL+19]. Für die funktionale Korrektheit der Echtzeiterweiterung wurden auch Vereinfachungen gegenüber der realen Architektur gemacht. Dies soll die Komplexität des Projektes gering halten und Teile des C-Quellcodes in der Validierung, die keine Beiträge zur Funkionalität leisten, werden nicht validiert. Der Maroon Planer könnte funktional vertifiziert werden, nach dem Abschluss dieses Projektes.

Außerhalb des Forschungsgebietes über Betriebssystemen wird der Coq Beweis-Assistent vielfältig angewendet. Es gibt einen Ansätze mit Hoare-Logik und SSREFELCT (s. [CZC+15] [IS14]). Die Funktionale Korrektheit eines Datei-Systemes konnte mit dem Coq-Beweisassistenten validiert werden. Diese Arbeiten zeigen gut, wie mithilfe der Grundlagen der Logik eine Vertifikation durchgeführt werden kann.

Der Beweis-Assistent Coq bietet auch die Möglichkeit Code zu exportieren (s. [ABB+17]). Dabei sei aber darauf hinzuweisen, dass im Rahmen des AQUAS-Projktes nicht die nötige Einarbeitungszeit für ein solches Projekt bestand und diese Idee nich weiter verfolgt wurde.

- besprechen: mit Kasier hier könnte ich ganz viel schreiben

Da Coq Plattform abhängig ist und es ständige Updates von PROSA gibt, müssen viele technische Probleme bewältig werden.

2.1.2.3 Zeitverhalten Verifikation

(> 2S)

⇒ Werner

2.1.3 Geschichteter Kernel

(10 S)

- Schichten auflisten wie im Antrag 3.1 – 3.4 -> Kaiser
- Analyse uK-Architekturen -> Züpke
- Schnittstellendefinitionen -> Züpke
- C-Impl. -> Züpke
- Akt. Masterprojekte, C nach Ada -> Dedi, Kaiser

2.1.4 Evaluation

(5 S)

<ToDo: (von unserem Ansatz, BMBF-Gliederung)>

Evaluation der Methode zur funktionalen Verifikation

Die in **??** beschreibene Methode zur statischen funktionalen Verifikation wurde in Teilen getestet und die Ergebnisse evaluiert. Am Beispiel einer Queue wurde ein SysML/OCL Modell erstellt und manuell in ein SPARK API und SPARK Contracts transformiert.

Aufwandsanalyse funktionaler Verifikation

- Ergebnisse Dedi Queues -> Dedi
  - Beschreibung des Queue-Interface in SysML/OCL
  - manuelle Transformation von SysML in SPARK API und von OCL in SPARK Contracts
  - Transformation Problemlos, das OCL noch mächtiger als SPARK
  - Verifikation des Interfaces ohne Implemenierung
  - durch hierarchische Schichtung Verifikation ohne Implementierung möglich

  - Postconditions der höheren Schichten können durch Deduktion aus Postconditions der niederen Schichten bewiesen werden.
- Ergebnisse Kopatschek -> Dedi
  - Implementierung des Marron-Schedulers in SPARK und Verifikation
  - Modularisierung C vs SPARK Code
  - Beispiele Vorgehen Verifikation
  - Simulation des Schedulers
  - Abdeckung Verifikation
  - ToDo: Tests für nciht verifizierten Code
  - Effort Analyse
  - Vergleich mit seL4
- Muen-Kernel Bomke -> Dedi
  - Beschreibung Muen-Kernel
  - Muen-Kernel ist frei von Laufzeitfehlern
  - Keine funktionale Verifikation bisher
  - Bounded Containers haben haben SPARK Spezifikation, aber Ada Body
  - Ziel: Beispielhaft Ada-Implemenierung eines Containers durch verifizierte SPARK-Implemeniterung austauschen
  - Bounded Vector Implementierung in SPARK umgesetzt
  - Abdeckung Verifikation
- Ggf. Akt. Masterprojekte? -> Kaiser
- Ergebnisse Werner -> Werner

2.1.5 AQUAS Kontext

2.1.5.1 SSP Methodik

(2 - 3S)

- Was will AQUAS -> Kaiser
- Interaction Points einführen -> Dedi
- Vorgehen in IP (aus dem D1.9) -> Dedi
- Einbettung HSRM Methode in AQUAS Methodik -> Dedi (Prio 2), Alt. Kaiser / Kröger

2.1.5.2 Anwendungsfälle

(1 S)

- Einführung, andere AF -> Werner
  - Wozu
  - Was gibt es, was soll nachgewiesen werden

2.1.5.3 ATM

(3 - 5 S)

⇒ Beckmann

- Ggf. IP??
- DDS
- Security (Angriffsszenarien)
- Beratung

2.1.5.4 Space Multicore

(8 - 10 S)

⇒ Werner

- Anwendung SSP Methode mit IP
- Eval RTEMS Funktionalität
- Portieren von RTEMS auf Plattform
- Ersetzen von RTEMS durch Marron + Adapter
- IPs
  - IP 1 Kopatschek
  - IP 3 Absint
  - IP 4 Performance-Messung
- Gap-Analyse D1.9 -> Werner

2.2 Wichtigste Positionen des zahlenmäßigen Nachweises

(1 S)

⇒ Beckmann, Gürtler

2.3 Notwendigkeit und Angemessenheit der geleisteten Arbeit

(0.5 S)

⇒ Beckmann (Entwurf)

2.4 Voraussichtlicher Nutzen

(1 S)

⇒ Beckmann (Entwurf)

2.5 Fortschritt bei anderen Stellen

(1 S)

⇒ Alle, insb. Dedi, Werner (State of the Art, max 1 Seite)

2.6 Erfolgte und geplante Veröffentlichungen

(2 S)

⇒ 1. Liste Beckmann

3 Erfolgskontrollbericht

3.1 Beitrag des Ergebnisses zu den förderpolitischen Zielen des Förderprogramms (Steigerung der Verbund-/Drittmittelfähigkeit

3.2 Wissenschaftlich-technische Ergebnisse des Vorhabens, erreichte Nebenergebnisse und gesammelte Erfahrungen

3.3 Fortschreibung des Verwertungsplans

3.3.1 Erfindungen/Schutzrechtsanmeldungen

3.3.2 Wirtschaftliche Erfolgsaussichten nach Projektende

3.3.3 Wissenschaftliche und/oder technische Erfolgsaussichten

3.3.4 Wissenschaftliche und wirtschaftliche Anschlussfähigkeit

3.4 Arbeiten, die zu keiner Lösung geführt haben

3.5 Präsentationsmöglichkeiten für mögliche Nutzer

Buchbeiträge

Tagungsbeiträge

Examensarbeiten

3.6 Einhaltung der Ausgaben- und Zeitplanung

Literaturverzeichnis

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1807–1823, New York, NY, USA, 2017. Association for Computing Machinery.

[BLDB06] Reinder Bril, Johan Lukkien, Rob Davis, and Alan Burns. Message response time analysis for ideal controller area network (CAN) refuted. *Information and Computation/information and Control - IANDC*, 01 2006.

[coq] The Coq Proof Assistant. Accessed: 2019-01-09.

[CSB16] F. Cerqueira, F. Stutz, and B. Brandenburg. Prosa: A case for readable mechanized schedulability analysis. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, ECRTS 2016, page 273–284, New York, NY, USA, 2016. Association for Computing Machinery.

[CZC+15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP 2015, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[GLL+19] Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. Integrating formal schedulability analysis into a verified os kernel. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 496–514, Cham, 2019. Springer International Publishing.

[IEC] Functional safety of electrical/electronic/programmable electronic safety-related systems.

[IS14] Aleksandar Nanevski Ilya Sergey. Introducing functional programmersto interactive theorem proving and program verification. Technical report, IMDEA Software Institute, 2014. acessed 2020-01-23.

[Isa] Isabelle. Accessed: 2020-01-21.

[Kai09] R. Kaiser. *Virtualisierung von Mehrprozessorsystemen mit Echtzeitanwendungen*. PhD thesis, Universität Koblenz-Landau, 02 2009.

[LNR+80] K.N. Levitt, P. Neumann, L. Robinson, United States. National Bureau of Standards, Institute for Computer Sciences, and Technology. *The SRI Hierarchical Development Methodology (HDM) and Its Application to the Development of Secure Software*. Number v. 13 in NBS special publication. U.S. Department of Commerce, National Bureau of Standards, 1980.

[ML08] Christian Schallhart Martin Leucker. A brief account of runtime verificationl. 2008.

[PdAC+19] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. Software foundations. In *Logical Foundations*, volume Volume 1 of *5*. 2019. found online at `https://softwarefoundations.cis.upenn.edu/current/lf-current/index.html` acessed 2019.

[PRO] Prosa - ECRTS'16 Artifact Evaluation. Accessed: 2020.

[spaa] Spark 2014. Accessed: 2019-12-03.

[spab] Subprogramm contracts in SPARK 2014. Accessed: 2019-12-05.

[ssr] Ssrefelct. Accessed: 2020-01-22.

[THW94] K. Tindell, H. Hansson, and A. J. Wellings. Analysing real-time communications: Controller area network (can). In *Proceedings of the 15th Real-Time Systems Symposium*, RTSS 94, 1994.